Towards a Standard PaaS Implementation API: A Generic Cloud Persistent-Storage API

Abstract: Platform as a Service (PaaS) supports application developers with the ability to implement and deploy their applications in the cloud. Several heterogeneous PaaS platforms are available, such as Google AppEngine (GAE), Windows Azure, Cloud Foundry, and OpenShift. Each PaaS provider has its own proprietary implementation and deployment APIs. The heterogeneity of these APIs makes developers worry about their application portability and interoperability. The work in this paper concerns about the heterogeneity of different PaaS implementation APIs. A standard PaaS implementation API, called Std-PaaS API, has been proposed to solve the application portability problem. Std-PaaS API allows developers to develop generic cloud application by writing their applications once and deploying many times on heterogeneous PaaS providers. Std-PaaS has been evaluated using a case study, in which a generic API for cloud persistent-storage service has been developed and used to write an application to upload files onto GAE and Windows Azure.

Keywords: PaaS; Vendor lock-in; Standard API

1 Introduction

Cloud computing has become very popular and has attracted many users to use it because of its economic and business benefits. Cloud computing provides everything as a service (XaaS). It allows users to provision multiple cloud resources in an elastic manner and to only pay for their actual usage. Cloud computing provides three service models, namely Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). The PaaS model provides application developers with APIs to develop, test, deploy, and manage their applications remotely in the cloud (Furht and Escalante, 2010; Buyya *et. al.*, 2011). This paper focuses on the PaaS model and more specially on application development.

The main advantage of PaaS service model is that cloud developers can concentrate on their major tasks without worrying about installation and maintenance of Integrated Development Environment (IDE) and data storage (Cloud4SOA, 2011; Zeginis *et. al.*, 2013). PaaS shifts the load of software installation and maintenance to PaaS cloud providers. Examples for currently available PaaS specific platforms are Google App Engine (GAE)[a] , Windows Azure[b] , OpenShift (OS)[c] , CloudFoundry (CF)[d] , and Amazon Web Service (AWS)[e].

Each PaaS platform provides its own proprietary APIs that are divided into two categories, implementation API and deployment API. A PaaS implementation API, also called development API, provides cloud developers with access and control of the PaaS platform resources, such as network, messaging, and data storage, which they typically use to develop cloud applications. A PaaS deployment API helps cloud developers to deploy and manage their applications on a specific cloud. Due to the heterogeneity of these APIs, a vendor lock-in problem faces current PaaS platforms.

The vendor lock-in problem makes cloud developers are unable to migrate their applications even if a PaaS cloud provider increases price, changes or violates the service level agreement, or removes some security policies (Roth, 2011). PaaS lock-in occurs at two levels, (1) application level, whereby migrating an application may require major refactoring and even complete rewriting of the application and (2) data level, whereby the data cannot be transferred between different PaaS platforms because each cloud provider has its own data format (Babar and Chauhan, 2011). The work in this paper provides a solution for application lock-in problem.

Several researches have dealt with the vendor lock-in problem and application portability as it is elaborated in the related work section. Sellami *et. al.* (2013) provided the COAPS API as a generic API for deployment and management of cloud applications. The COAPS API allows cloud developers to deploy and manage their applications on CloudFoundary and OpenShift. In our previous work, we have extended the COAPS API to allow deployment and management on GAE platform (Hossny *et. al.*, 2013, In Press). However, the COAPS API was concentrated only on deployment APIs and ignored implementation APIs. Rafique *et. al.* (2014) proposed an abstraction layer to allow portability and interoperability for hybrid cloud applications. They have created a uniform API for two service, storage service and asynchronous tasks. Their work is very similar to ours. However, their uniform API is not an open source. Also, the evaluation of their architecture leads to extra overhead. To the best of our knowledge, currently, there is no established standard or generic implementation API for developing cloud applications.

In this paper, we propose a Standard PaaS Implementation API (called Std-PaaS API) that is used by application developers to develop their cloud applications in a generic way independent of specific PaaS platforms. By using the proposed Std-PaaS API, application developers implement their applications once and deploy many times (i.e., deploy the same application on multiple heterogeneous PaaS platforms). In a sense, the proposed Std-PaaS API provides a middleware layer to communicate between a set of heterogeneous PaaS platforms and cloud application developers. A generic API for cloud persistent storage has been designed, as a case study, to evaluate the proposed Std-PaaS API. This generic API includes two adapters, GAE and Windows Azure. It has been tested by developing one generic application that is used to upload files either onto GAE or Azure. This generic application can be deployed using the COAPS API. Figure 1 clarifies the relationship between the Std-PaaS API and the COAPS API.

The rest of the paper is structured as follows. Section 2 provides detailed description of the design of the proposed standard PaaS implementation API (Std-PaaS API) and its architecture. The implementation details and evaluation are provided in Section 3. Section 4 discusses related work. Section 5 concludes the paper and presents future work.

2 Standard PaaS Implementation API

In this section, we elaborate the design and the architecture of Std-PaaS implementation API. Also, a detailed description of how to design a generic API inside the Std-PaaS API is included. As an example, this generic API is implemented for persistent storage service.

[a] http://code.google.com/appengine
[b] www.microsoft.com/azure
[c] https://www.openshift.com/
[d] http://www.cloudfoundry.com/
[e] http://aws.amazon.com/

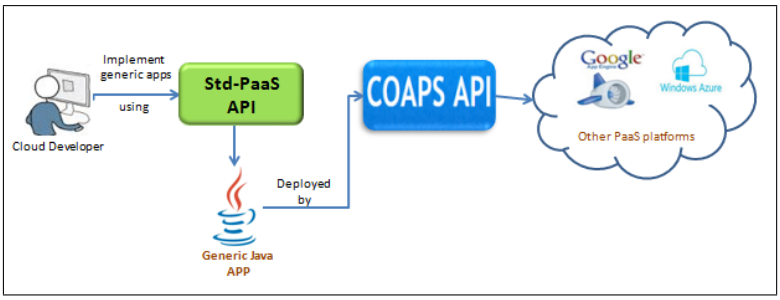

Figure 1 The relationship between the Std-PaaS API and the COAPS API

A Standard PaaS Implementation API (called Std-PaaS API) is proposed to overcome the vendor lock-in problem by providing a set of standard service implementation APIs to cloud application developers to help them in implementing their cloud applications in a generic fashion. Std-PaaS API works as a middleware layer to communicate between heterogeneous PaaS platforms and cloud application developers. Std-PaaS API architecture is illustrated in Figure 2. It is composed from three main layers, Std-PaaS API layer (bottom), adapters layer, and specific PaaS APIs layer (top). The Std-PaaS API layer provides cloud developers with a set of generic methods to allow them to develop generic applications. The adapters layer provides an adapter for each specific PaaS platform to map the Std-PaaS API generic methods into the specific PaaS API methods. The specific PaaS APIs layer provides the specific PaaS API of each PaaS platform. Extendability is considered one of the main advantage of Std-PaaS API architecture; where it can be extended to include other PaaS platforms by only implementing their adapters.

For a concrete example, we have designed a generic API for the cloud Persistent Storage Service (PSS). The PSS helps cloud developers to store large files remotely in the cloud. These large files are called BLOBs (Binary Large OBjects) or objects and it can be of any type and up to terabytes in size. These blobs are stored inside containers (or buckets). Each PaaS provider supports developers with a proprietary API for PSS; for example, GAE provides Google Cloud Storage (GCS) API (Google, 2016) and Azure provides the Azure Blob Storage API (Microsoft, 2016).

To design a generic PSS API, we adopted the following methodology, which is described in more details in the next subsections. First, we analyzed the vendor-specific PSS API for PaaS platforms; we took GAE and Azure as examples as they are widely used. Second, we compared the specific APIs and identified a set of semantically common features among them as well as a set of unique features for each API. Finally, we designed the generic API based on the common features.

2.1 Studying Specific Persistent Storage Service (PSS) API

The PSS API of both GAE and Azure have been studied and tested. Google Cloud Storage (GCS) (Google, 2016) allows users to store and retrieve large files at any time. These files are called objects and they must be stored in buckets that must belong to a project. Buckets cannot be nested and can be used to organize files and control their access. Also, buckets cannot be shared between projects. A project can contain several buckets. Table 1 provides a summary of GAE PSS and the role of each API. GAE PSS provides several main APIs which specified as the following. *GcsService*: this interface is used to create and access files in GCS; *GcsFilename*: this class creates a GCSFile with a given bucket and a file name;

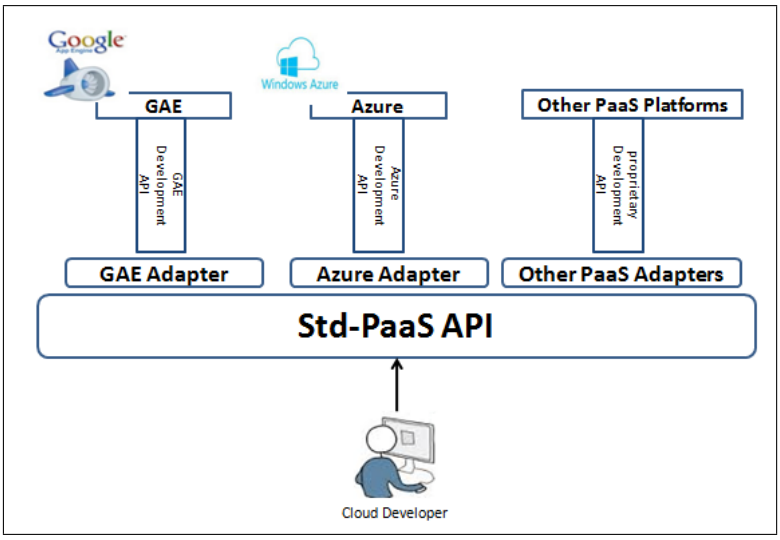

Figure 2 Std-PaaS API Architecture

GcsFileOptions: this class is used to store options (e.g., Mime Type and ACL) for creating Google storage files; and *GcsOutputChannel*: this interface provides output channel to write data to GCS.

PSS for Azure is provided through Azure Blob Storage (Microsoft, 2016). It allows users to store and retrieve large files up to a terabyte. These files are called blobs, which must be stored in containers that must belong to a storage account. Azure supports two types of blobs, namely block blob and page blob. The block blob has a maximum size of up to 200 GB, whereas the page blob has a maximum size of up to one TB. A storage account can contain several containers. Table 2 provides a summary of Azure PSS and the role of each API. Azure PSS provides several main APIs which specified as the following. *CloudStorageAccount*: this class is used to create a Windows Azure Storage account using an account name and key; *CloudBlobClient*: this class is used to create a client, of the Azure blob service, that is used to execute requests against the Azure blob service; *CloudBlobContainer*: this class is used to create a container in the Azure blob service; *BlobContainerPermissions*: this class is used to update a container permission; and *CloudBlockBlob*: this class represents a blob that can be uploaded as a set of blocks.

2.2 Comparison of GAE PSS API and Azure PSS API

A comparison between GAE and Azure PSS APIs has been done. This comparison is based on identifying a set of semantically common features between both GAE and Azure and identifying a set of unique features for each PaaS platform.

Typical usage scenarios of both GAE and Azure PSS APIs involve the following five semantic steps. (1) ***Configuration*** is used to get from a user the required parameters to execute a specific PSS API; (2) ***Create Container*** is used to create a container if possible; (3) ***Create Blob*** is used to create a blob inside a container; (4) ***Upload Blob*** is used to upload a given input stream into the created blob; and (5) ***Access Blob*** is used to access the uploaded blob.

Table 1 GAE Persistent Storage Service (PSS) API

GAE PSS API (GCS)	Its role
Create instance of *GcsService*	This instance will be used to create a blob.
Create instance of *GcsFilename*.	This instance will be used by GCSService instance to create a file. This is done using a bucket and a file name.
Create instance of *GcsFileOptions*	It is used to set file permissions and a content type.
Create instance of *GcsOutputChannel*.	It is used to open an output channel to a file to allow writing on it. It is done using the created GcsFileName instance and the created GcsFileOptions instance. It will create the file if it does not exist; otherwise, it will replace it with the new one.
Create instance of *OutputStream*.	This output stream will be used to write data to a file. This is done using *Channels.newOutputStream* Java method which takes the created GcsOutputChannel instance as a given input .
Implement *copy*(*InputStream*, *OutputStream*).	The copy method takes the input stream of a given file and the output stream of the previously created channel. It is used to copy data from the input stream of a given file to the output stream of the created channel.

Table 2 Azure Persistent Storage Service (PSS) API

Azure PSS API	Its role
Create instance of *CloudStorageAccount*.	This storage account is used to create *CloudBlobClient*. This is done using *CloudStorageAccount.parse* method which takes, as a given input, a connection string that is containing an account name and key.
Create instance of *CloudBlobClient*.	This blob client is used to get a reference to a container. This is done by calling the *createCloudBlobClient* method of the created CloudStorageAccount.
Create instance of *CloudBlobContainer*.	This container can be used to optionally update its permission and get a reference to a blob. This is done by calling the *getContainerReference* method of the created CloudBlobClient.
Create instance of *BlobContainerPermissions*	This is used to change a container permission to be publically accessible. This step is optional, whereby the container has a default permission of "private access".
Create an instance of *CloudBlockBlob*.	This instance is used to upload a blob as a set of blocks. This is done by calling *getBlockBlobReference* method of the created CloudBlobContainer.
Call the *upload(FileInputStream, FileLength)* method of *CloudBlockBlob* instance.	The upload method takes the FileInputStream of a given file and its length. It is used to upload data from a given input stream to the created CloudBlockBlob.

These common steps hide many implementation-specific details. For instance, (1) a stored file in GAE is called an object, whereas in Azure it is called a blob. Therefore, object and blob are semantically equivalent; (2) Azure stores blobs in containers, whereas GAE stores them in buckets. Therefore, containers and buckets are semantically equivalent; (3) GAE requires using an existing container, whereas Azure creates a container if it does not exist; (4) Azure requires a storage account (with an account name and a key) to access its persistent storage service, whereas GAE requires a project to access its persistent storage service; (5) Azure provides a method to update container permissions, whereas GAE does not support this option; and (6) GAE requires the file content type of an uploaded file, whereas Azure does not require the file content type. Table 3 summarizes how the five generic steps are implemented in each specific PSS.

Table 3 Generic Steps of the Std-PaaS persistent-storage API *vs*. specific API steps.

Specific PSS API / Generic Step	Azure Blob Storage (Microsoft, 2016)	Google Cloud Storage (GCS) (Google, 2016)
Configuration	The configuration parameters include the Azure storage account with an account name and a key, a container name, which will be created if it does not exist, and an input stream for a file to be uploaded.	The configuration parameters include a container name, which must be created a priori; otherwise an exception is raised and an input stream for a file to be uploaded.
Create Container	Create the following instances: a cloud storage account, a cloud blob client, and a cloud blob container.	The container must be created from Google developer console before the application starts. This is because no API is available for creating a container. So, no container establishment is needed from inside an application.
Create Blob	Create an instance of CloudBlockBlob using the previously created container.	Create the following instances: GcsFilename to specify a bucket and a file name, GcsFileOptions to specify a file content type and its Access Control List (ACL), and GcsService to use the previously created instances to create or replace the given file; the return value is a GcsOutputChannel instance.
Upload Blob	Call the upload method of the previously created CloudBlockBlob instance and give it an input stream of a given file. A URI of the uploaded file is returned, which follows the template: http://<your_Azure_storage_service>.blob.core.windows.net/<container_name>/<file_name> For example: http://myteststore3.blob.core.windows.net/mycontainer/capture.png	No direct method to do the upload. So, we need to copy data from a source to a destination using a buffer for example. A URI of the uploaded file is returned, which follows the template: http://storage.googleapis.com/<bucket_name>/path/to/object For example: http://storage.googleapis.com/my_bucket_test/Capture.PNG
Access Blob	Using the previously returned URI.	Using the previously returned URI.

3 Implementation and Evaluation

In this section, we represent the implementation details of the Std-PaaS persistent storage API (i.e., the implementation details of the proposed generic five steps), a case study is provided to evaluate this API on both GAE and Azure, and finally, some snapshots are depicted to illustrate the deployed application on GAE and Azure.

As stated previously, one of the advantages of Std-PaaS API is that it is extendible and can support multiple heterogeneous PaaS platforms. In addition, the Std-PaaS API is available as an open source and can be downloaded from (Std-PaaS4BlobStorage, 2016). In the current state, it supports GAE and Azure. We have implemented two adapters for Std-PaaS persistent-storage API, namely GAE Adapter and Windows Azure Adapter. They are implemented in JAVA. Each adapter maps the Std-PaaS persistent-storage API into a specific PaaS provider API. To extend the Std-PaaS persistent storage API with a new PaaS platform, only a new adapter should be implemented.

The methods of the Std-PaaS persistent-storage API are based on the generic steps identified in the previous section and are listed in Figure 3. These methods include: (1) *configureStorageService*, which takes an XML manifest as an input (a sample of this manifest file is depicted in Figure 4) and parses it to get the configuration parameters. These parameters are then stored in an object of class *StorageServiceType*. The StorageServiceType class keeps states of our Std-PaaS PSS API. In other words, an instance of StorageServiceType is passed among the generic methods of the Std-PaaS PSS API to store all required parameters. By this way, the StorageServiceType class makes the design of our Std-PaaS PSS API is stateful. A snapshot of the StorageServiceType's parameters is clarified in Figure 5; (2) *createContainer*, which takes an object of StorageServiceType (contains the configuration parameters) as an input. It creates a container using the specific PaaS API. Finally, it stores the created container object in the given StorageServiceType object to be used by the next functions; (3) *createBlob*, which takes an object of StorageServiceType (contains the container object) and uses the given container object to create a blob object. Finally, it stores the created blob object in the given

StorageServiceType object to be used by the next functions; (4) *uploadBlob*, which takes an object of StorageServiceType (contains the blob object) and uses the given blob object to upload a given input stream. Finally, it stores the URL of an uploaded blob in the given StorageServiceType object to be used by the next function; and (5) *accessBlob*, which takes an object of StorageServiceType (contains a URL of an uploaded blob) and uses the given URL to access a blob.

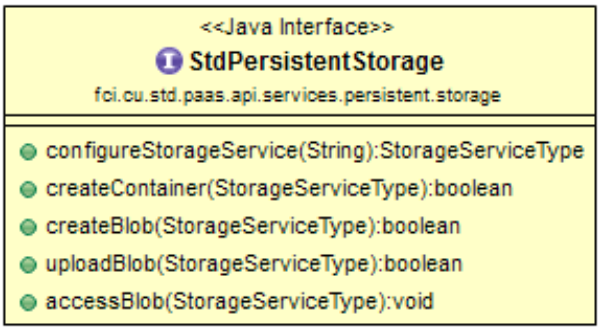

Figure 3 Std-PaaS Persistent Storage API Methods

```
<persistent_storage_service_manifest>
  <description>
     this manifest describes a sample of a persistent storage service manifest
  </description>
  <storage_service_account>
        <account_name>myteststore3</account_name>
         <account_key>secret account key </account_key>
   </storage_service_account>
   <container name="mycontainer"   provider="Azure"/>
</persistent_storage_service_manifest>
```

Figure 4 A Sample Persistent Storage Service Manifest

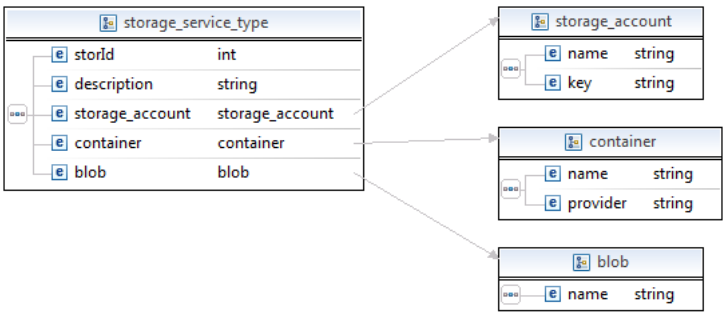

Figure 5 The required parameters of StorageServiceType class

3.1 Implementation Scenario using the Std-PaaS PSS API

The Std-PaaS PSS API helps cloud developers to implement generic cloud JAVA applications which uploads files to a cloud persistent storage. In the current state, these implemented applications can be deployed on both GAE and Azure. The required steps to implement such generic applications using the proposed Std-PaaS PSS API are:

- Get from a user an input stream, which needs to be uploaded, and an XML manifest which contains information such as the one clarified in Figure 4.

- Call from the Std-PaaS PSS API the following methods respectively:

  – *configureStorageService* method which takes, as an input, the given XML manifest. It parses the manifest and stores the parsed data in an instance of *StorageServiceType* class. Finally, the StorageServiceType instance is returned as an output.

  – *createContainer* method takes, as an input, the previously returned StorageServiceType instance and creates a container using a specific PaaS API and the parameters defined in the given StorageServiceType instance. The created container instance is stored in the given StorageServiceType instance. Finally, this instance is returned as an output.

  – *createBlob* method takes, as an input, the previously returned StorageServiceType instance and creates a blob using a specific PaaS API and the previously created container. The created blob is stored in the given StorageServiceType instance. Finally, this instance is returned as an output.

  – *uploadBlob* method takes, as an input, the previously returned StorageServiceType instance and uploads the given input stream using a specific PaaS API and the previously created blob. A URL of the uploaded blob is stored in the given StorageServiceType instance. Finally, this instance is returned as an output.

  – *accessBlob* method takes, as an input, the previously returned StorageServiceType instance and uses the URL of an uploaded blob to access it.

A cloud developer can follow the previous scenario to implement generic cloud java applications and deploy these applications many times on multiple heterogeneous PaaS platforms as long as they are supported by the Std-PaaS PSS API. To deploy these applications on a specific PaaS platform, a cloud developer only needs to import its specific adapter jar library into the application; for example, to deploy a generic application on GAE, the GAE adapter is imported in this application.

From our case study, Figure 6 and Figure 7 depict snapshots of a deployed application which takes an XML manifest and an input stream as inputs. The XML manifest contains a set of required configuration parameters (e.g., account name and account key in case of Azure). The input stream represents the file that will be uploaded to a cloud. If the uploading process succeeds, a URL of the uploaded file is returned to be used by the user to access the file. Figure 8 presents a snapshot of the returned response when uploading to Azure storage and same thing will happen when uploading to GAE storage.

Figure 6 The case study Java application while uploading a file to Google cloud storage using Std-PaaS API.

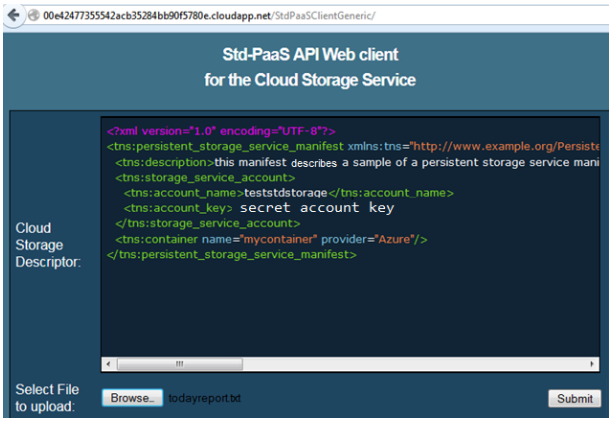

Figure 7 The case study Java application while uploading a file into Azure cloud storage using Std-PaaS API.

Figure 8 The response after uploading a file to Azure storage.

4 Related Work

Previous research has dealt with the vendor lock-in problem and application portability. However, still the problem persists and no complete solution to this problem is available till now. The authors in (Escalera and Chavez, 2012) created a UML model for common services among different PaaS providers. This UML model can be used by PaaS providers to develop their APIs in a homogeneous way. Therefore, their solution is idealistic. However, our approach is more pragmatic.

Cretella and Di Martino (2012) handled application portability by semantically analyzing the provider APIs to understand their functionalities and to create portable cloud

applications. They proposed automatic alignment technique for API analysis and automatic technique for API annotation. Their API alignment technique requires manual validation. Therefore, it is not fully automatic. Also, they proposed a semantic engine (Cretella and Di Martino, 2014) to help cloud developers in identifying the APIs that are required to develop portable cloud applications. Although their approach helps cloud developers in specifying the data flow of several connected services and facilitate the alignment of different services, it cannot handle problems related to data portability and interoperability.

Androcec and Vrcek (2012) tried to solve application portability issues by designing a PaaS API ontology to identify the common concepts among PaaS platforms APIs. Their ontology is very simple and it is not used practically in, at least, a case study. Based on our analysis, their ontology defines several concepts in a high level of abstraction which may require more update and enhancement when it is used in practical situations. Paraiso *et. al.* (2014) provided a multi-cloud computing PaaS solution. Their multi-cloud solution allows different and independent cloud providers to interoperate with each other. Their solution is called soCloud PaaS which helps developers in managing portability and deploying their applications across multiple clouds. The soCloud is restricted to only deploying service oriented based applications.

Rafique *et. al.* (2014) proposed a middleware layer (i.e., abstraction layer) to support portability and interoperability for hybrid cloud applications. They have created a uniform API for storage service and asynchronous tasks service. Their uniform API supports GAE and OpenShift. OpenShift did not provide a blob storage API, while GAE has its own blob storage API. So, it is not clear how the authors build their blob storage uniform API for both GAE and OpenShift. Although, their work is very similar to ours, there are critical differences with our work. The main differences are (i) their uniform API supports both GAE and OpenShift, whereas our Std-PaaS generic API supports both GAE and Azure, (ii) the evaluation of their middleware leads to extra overhead, and (iii) their uniform API is not an open source.

5 Conclusions and Future Work

In summary, the proposed Std-PaaS implementation API aims to overcome the vendor lock-in problem at the application level. It can be used by application developers for implementing generic cloud applications that are independent of any specific PaaS platform, that is, portable applications. As a first step, we have designed a Std-PaaS persistent storage API with two adapters for GAE and Azure. As a case study, we have implemented a JAVA application that uses the proposed API to implement a Web client to upload files to either Google cloud storage or Azure storage based on a user-supplied manifest file.

Since the created adapters depend on each specific provider API, any change or update in these specific APIs is more likely to affect the adapters and may require refactoring or complete rewriting. Therefore, as an ongoing work, we started to use semantic ontology to generate the adapters in an automatic or semi-automatic way. Also, we are planning to enrich the Std-PaaS implementation API with more services API besides the persistent storage service API.

References

Furht, B. and Escalante, A. (2010) 'Handbook of cloud computing', Springer-Verlag New York Inc.

Buyya, R., Broberg, J., and Goscinski, A. (2011) 'Cloud Computing Principles and Paradigms', Wiley Press, New York, USA.

Cloud4SOA (2011) 'D1.2 Cloud Semantic Interoperability Framework', Technical report. [online]http://www.cloud4soa.com/sites/default/files/D1.2_Cloud4SOA%20Cloud%20Semantic%20Interoperability%20Framework.pdf (Accessed 1 Jan. 2016).

Zeginis, D., D'Andria, F., Bocconi,S. , Gorronogoitia Cruz, J., Collell Martin, O., Gouvas, P., Ledakis, G., and Tarabanis, K. a. (2013) 'A user-centric multi-PaaS application management solution for hybrid multi-Cloud scenarios', *Scalable Computing: Practice and Experience*, Vol. 14, No. 1, pp. 17–32.

Roth, I. (2011) 'Cloud Got You Locked-in? Avoid it by Choosing an Open PaaS' [online]https://www.openshift.com/blogs/cloud-got-you-locked-in-avoid-it-by-choosing-an-open-paas (Accessed 1 Jan. 2016).

Babar, M. A. and Chauhan, M. A. (2011) 'A Tale of Migration to Cloud Computing for Sharing Experiences and Observations' in *SECLOUD 11. Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, ACM Conference on Computer-Human Interaction, pp. 50–56.

Sellami, M., Yangui, S., Mohamed, M., and Tata, S. (2013), 'PaaS-independent Provisioning and Management of Applications in the Cloud' in *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, Washington, DC, USA.

Hossny, E., Khattab, S., Omara, F., and Hassan, H. (2013) 'A Case Study for Deploying Applications on Heterogeneous PaaS Platforms', in *Proceedings of the 2013 International Conference on Cloud Computing and Big Data (CloudCom-Asia)*, FuZhou, China.

Hossny, E., Khattab, S., Omara, F., and Hassan, H. (In Press) 'Implementing Generic PaaS Deployment API : Repackaging and Deploying Applications on Heterogeneous PaaS Platforms', *International Journal of Big Data Intelligence*.

Rafique,A., Walraven, S., Lagaisse, B., Desair, T., and Joosen, W. (2014) 'Towards portability and interoperability support in middleware for hybrid clouds', in *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 7–12.

Google. (2016) 'What is Google Cloud Storage?' [online]https://cloud.google.com/storage/docs/overview (Accessed 2 Jan. 2016).

Microsoft. (2016) 'Introduction to Microsoft Azure Storage' [online]http://azure.microsoft.com/en-us/documentation/articles/storage-introduction/ (Accessed 2 Jan. 2016).

Std-PaaS4BlobStorage. (2016) Std-PaaS Generic API source code. [online]https://github.com/emanhossny/Std-PaaS4BlobStorage (Accessed 7 Jan. 2016).

Escalera, M.F.P. and Chavez, M.A.L. (2012) 'UML model of a standard API for cloud computing application development', in *9th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*, pp. 1– 8.

Cretella, G. and Di Martino, B. (2012) 'Towards Automatic Analysis of Cloud Vendors APIs for Supporting Cloud Application Portability', in *Sixth International Conference on Complex, Intelligent, and Software Intensive Systems*, pp. 61–6.

Cretella, G. and Di Martino, B. (2014) 'A semantic engine for porting applications to the cloud and among clouds', *Software: Practice and Experience*, DOI: 10.1002/spe.2304.

Androcec, D. and Vrcek, N. (2012) 'Platform as a Service API Ontology', in *Proceedings of the 12th European Conference on eGovernment*, pp. 47–54, Barcelona, Spain.

Paraiso, F., Merle, P., and Seinturier, L. (2014), 'soCloud: A service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds', *Springer Computing Journal*, pp.1–27, DOI: 10.1007/s00607-014-0421-x.